COMP-208: Computers in Engineering

Fortran, C and Algorithms
McGill University School of Computer Science
Winter 2006

Contents

Preface........ 1
I Introduction........ 3
II Basic Computer Concepts........ 9
III Starting to Program in Fortran........ 12
IV More Complicated Fortran Programming........ 14
V Advanced Fortran Features........ 15
VI Transition from Fortran to C........ 18
VII Some More of C, Using Standard Libraries........ 24
VIII Searching........ 26
IX Sorting........ 31
X Recursion........ 33
XI Merge Sort........ 38
XII Root Finding, Numerical Differentiation........ 41
XIII Numerical Integration........ 45
XIV Initial Value Problem........ 48
XV Linear Algebra........ 50
Appendix A Numbering Systems........ 58
Appendix B Data Representation........ 60
Appendix C Useful References........ 62

Preface

What's this, then?

This document was written for the COMP-208 Computers in Engineering class at McGill University; it aims at being at the same time an outline of what is discussed in lectures, class notes on the more difficult subjects, a list of supplemental material and outside references, and a general programming handbook. It is not meant as a substitute for lectures or tutorials but rather as a help to these. It's the kind of document that's a nice-to-have on paper during lectures because students can read-along and take notes, it's also a very good reference for assignments, and since it's a PDF document it's searchable and available anywhere computers are. Tree lovers might consider bringing a laptop to class and taking notes on their favorite word processor.

This document also comes with quite a bit of code which is available online; parts of this code is replicated in the document for explanation purposes.

On Evolution and Intelligent Design

This document will probably never be finished: the course evolves from year to year according to students' feedback. It is meant to evolve throughout each session, therefore if there is anything that is unclear, if there are typos, blatant errors, outright lies, unsightly formatting bugs, or if you simply would like to know more on any of the subjects covered in this class please feel free to email the course staff and it will be our pleasure to make this document better.

Who

This document was put into form by Jean-François Bastien for the Fall 2005 session. It is a collection, expansion and reorganization of material from two previous lecturers — Gerald Ratzer and Olivier Giroux — which also contains material developed by said author during his own service as a teacher assistant for this class.

Formatting

Mostly all the text in this document is set in Bitstream Vera Serif and code is in Courier New. When code is meant to be replaced it is in red and italic. To mimic code editors in long code samples line numbers are included on the left margin; language keywords are in blue and bold; preprocessor commands are in yellow; strings are in green and italic; and comments are in Bitstream Vera Sans green italic.

Copyrights

I Introduction

What is this class?

Official description

- Introduction to computer systems. Concepts and structures for high level programming. Elements of structured programming using Fortran 90 and C. Numerical algorithms such as root finding, numerical integration and differential equations. Non-numerical algorithms for sorting and searching
- Three hours of lecture per week
- Prerequisite: differential and integral calculus
- Co-requisite: linear algebra (determinants, vectors, matrix operations)

More precisely

Preface
I Introduction
II Basic Computer Concepts
III Starting to Program in Fortran
IV More Complicated Fortran Programming
V Advanced Fortran Features
VI Transition from Fortran to C
VII Some More of C, Using Standard Libraries
VIII Searching
IX Sorting
X Recursion
XI Merge Sort
XII Root Finding, Numerical Differentiation
XIII Numerical Integration
XIV Initial Value Problem
XV Linear Algebra
Appendix A Numbering Systems
Appendix B Data Representation
Appendix C Useful References

Why does this class exist?

- A lot of engineering starts off with theory which is then implemented as software: the problem is solved by a computer
- Engineers often use software without writing it, but they need to be able to: understand the results, criticize them, understand the software's limitations, extrapolate from the results
- Engineers sometimes write software
- Other core engineering classes ask students to write programs
- Many assignments from other classes will be much easier to do or verify with programs

Why Fortran and C?

Fortran was designed in the 1950s for scientific calculations, and lots of legacy code is still written in it (including some that is taught at McGill). It is relatively simple to learn and students tend to prefer Fortran to C in this class.
C was designed in the 1970s to write operating systems, it is very efficient and powerful, and mostly all programming languages take from C in some way.
These two languages offer a good base and teach good practices in programming and allow students to learn other languages easily, either for engineering or computer science purposes.
For more on these two languages, see:

- http://en.wikipedia.org/wiki/Fortran
- http://en.wikipedia.org/wiki/C_Programming_Language

Who is giving this class?

Lecturers

Nathan Friedman	nathan@cs.mcgill.ca	(514) 398-7076	McConnell room 318

Teacher assistants

Jean-François Bastien	jean-francois.bastien@mcgill.ca
Guillaume Charpenel	guillaume.charpenel@mcgill.ca
Nicolas Gervasi	nicholas.gervasi@mail.mcgill.ca
Zouhair Mahboubi	zouhair.mahboubi@mail.mcgill.ca

Where and when is this class given?

Tuesdays and Thursdays from 2:35 to 3:55 PM in BURN 1B45 (Burnside Hall).

Grading

20%	Six assignments in total: three in Fortran, three in C
30%	Midterm exam
50%	Final exam

Tutorials

Tutorial attendance is not mandatory but is strongly recommended. People who have never programmed should definitely attend them, people who have already programmed can skip some of the tutorials but should not complain if they miss out on some of the more advanced tutorials which happen to be incredibly useful when doing the assignments.

Tutorial times are determined from an in class-poll on the first lecture to optimize student availability.

Resources

This document is available on WebCT (http://www.mcgill.ca/webct/) in PDF format. It is meant as a “read along and take notes on it” kind of thing, it is therefore recommended that it be brought to class either on paper or electronically.

Other resources for this class, in the order in which you should be consulting them:

- Going to classes, taking notes, listening and asking questions
- Going to tutorials, taking notes, listening and asking questions
- Going to TAs' office hours (to be posted on WebCT)
- Asking questions on WebCT's discussion forums (and reading previously answered questions)
- Searching on the Internet
- Sending an email to TAs or lecturers

There also is a lot of code available on WebCT, some of it is replicated in the document. Some of the code written in class or in tutorials will also be posted on WebCT.

McGill computer facilities

Lots of computers are available around campus, the engineering computers have all the software required for this course. The bigger computer facilities are at Frank Dawson Adams 1 (FDA 1) and Macdonald-Harrington G15 (MDHAR G15).

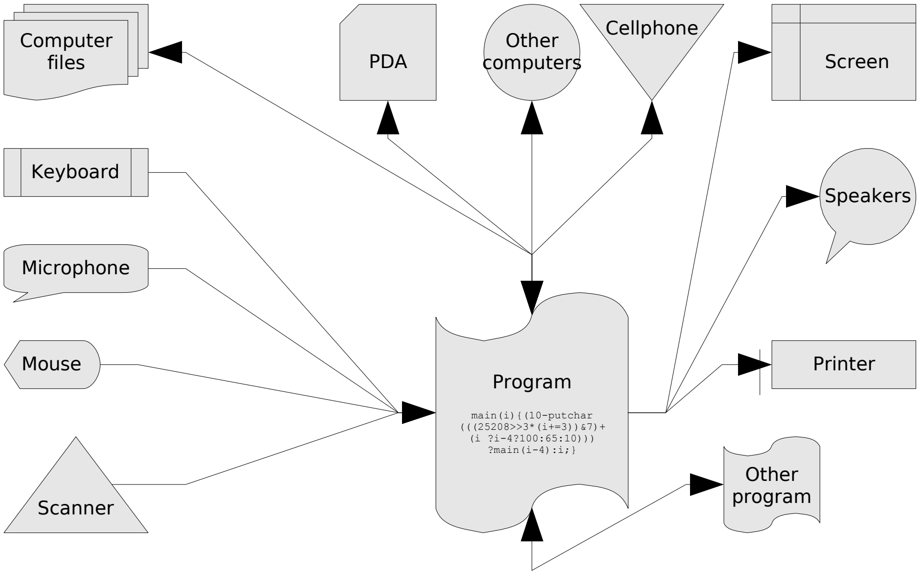
History

See:

- http://en.wikipedia.org/wiki/Programming_language
- http://en.wikipedia.org/wiki/Timeline_of_computing
- http://www.oreilly.com/news/graphics/prog_lang_poster.pdf

What is a program?

- A program (or software) tells a computer what actions or computations to carry out
- Computer architecture
- Operating system
- Program and data are:
  1. on the hard drive
  2. Loaded in memory
  3. Sent to the CPU and others
- Running a program
- Program interactions, some possible inputs and outputs of information:



```
main(i){(10-putchar
(((25208>>3*(i+=3))&7)+
(i ?i-4?100:65:10)))
?main(i-4):i;}
```

- Interactions that we will use in this class

Computer Languages, Source Code, Compilers

A computer thinks in what's called machine code which is binary — zeros and ones — and humans don't. It is possible to write a program in binary but aside from it being very impractical and hardly understandable, such a thing would be very hardware dependent: different machines “speak” a different kind of machine code. This is why computer languages (like Fortran and C) exist: a computer language is a human readable text that will be translated into machine code by a compiler.

More precisely source code is what's written by a human in a programming language, this mostly always uses English keywords to allow the programmer to tell the computer what to do. Computer languages, as will be seen later, force the programmer to adhere to certain rules of syntax, semantic and flow.

For the computer to be able to execute the commands given to it by the programmer in the source code there needs to be a translation from source code to machine code: that translator is called a compiler. Explained rapidly, the compiler reads the code and then checks to see if it's written in a well formed programming language and if it's error free, if so it translates that source code into an executable.

So, how do we go about writing source code then? Not in Word or anything like this:

source code must be written in a plain text editor, that is without any of that fancy formatting such as bold, italic, text alignment, font and images. This allows the programmer to concentrate on the contents of the code and not on its general looks: the looks of source code is insignificant to the computer and can be left very plain or can be automatically handled by text editors that were created precisely to edit code.
What's a binary exactly? It's usually a file whose extension is `.exe`. Therefore source code only needs to be compiled once and can then be run multiple times, but if changes are made to the source code then it needs to be recompiled for the executable to have these new instructions that were added to the source code.

Software to install

The software used in this class is already installed on the engineering computer facilities' computers; while students can work solely from the University's computers it is strongly recommended that they install a compiler and text editor on their own machine so that they may work from home.
Outlined below is a list of *suggested* software to use, it is by no means the only compilers and editors that will do the job well. The COMP-208 staff will provide some help with installing the software below if students encounter any trouble installing or using it.

For Windows XP, NT, 2000 users

1. Download the `software-Windows.zip` file from WebCT
2. Unzip the file
3. Run the `MinGW-5.0.0.exe` program and install the release version of `g77` and `g++`, you might need to change download servers if the one that's pre-selected isn't working
4. Copy the `scite` folder to `C:\Program Files\`
5. Make a shortcut to `SciTE.exe` and place it on your desktop or start menu (or wherever else you want it), this is your text editor

You can now use your command line interface by going to:
Start, Run..., Open: `cmd`, Ok.

For Windows Me, 98, 95 users

1. Download the `software-Windows.zip` file from WebCT
2. Unzip the file
3. Run the `MinGW-5.0.0.exe` program and install the release version of `g77` and `g++`, you might need to change download servers if the one that's pre-selected isn't working
4. Reboot your computer
5. Copy the `scite` folder to `C:\Program Files\`
6. Make a shortcut to `SciTE.exe` and place it on your desktop or start menu (or wherever else you want it), this is your text editor

You can now use your command line interface by going to:
Start, Run..., Open: `command`, Ok.

For Mac OS X users

1. Download the `software-Mac-OS-X.zip` file from WebCT
2. Unzip the file
3. Run the `cctools-576.dmg` program to install a few required programs
4. Run the `gfortran-macosx.dmg` program to install the `gfortran` and `gcc` compilers
5. If it isn't already installed, download and install Xcode from http://developer.apple.com/tools/xcode/, this is your text editor

You can now use your command line interface by going to:
Applications, Utilities, terminal.

For users of other operating systems

If you use Linux, some UNIX version, some BSD version, Solaris, Amiga, NeXT, GNU Hurd, or any other exotic operating system then we're assuming that you know how to install a compiler for Fortran and C, and how to get a text editor and a command line interface. If not, the COMP-208 staff can try to help you.

Other software that could be used

Although the COMP-208 staff will **not** provide support for the following software, it could nonetheless be used and might even be better that the above recommended software.

- Notepad for source code editing on Windows, this really isn't recommended
- GNU Emacs for Windows for source code editing, from http://www.gnu.org/software/emacs/windows/
- SciTE on Mac OS X, from http://www.scintilla.org/SciTE.html
- Microsoft Visual Studio, a (costly) Integrated Development Environment (IDE)
- Dev-C++, an open-source IDE, from http://www.bloodshed.net/devcpp.html
- Code::Blocks, another open-source IDE, from http://www.codeblocks.org/
- CDT, another open-source IDE, from http://www.eclipse.org/cdt/
- Microsoft Visual C++ Toolkit, a command line compiler, from http://msdn.microsoft.com/visualc/vctoolkit2003/
- Intel compilers (including Fortran and C), costly but good compilers, from http://www.intel.com/cd/software/products/asmo-na/eng/compilers/index.htm
- Code Warrior for Macintosh, from http://www.metrowerks.com/

If you would like to recommend other software, do not hesitate to contact the course staff.

II Basic Computer Concepts

What is a program from a programmer's point of view?

- Step by step guide that a computer follows to accomplish a task
- A computer is very rapid and executes the tasks very fast
- A computer isn't intelligent: it has to be told exactly what to do, if not it will merely speed and automate errors

This concept is crucial in this class: we will learn how to tell a computer to do our bidding. To do so, we must first learn what a computer can and can't do, and how to tell it to do something.

Paradigms

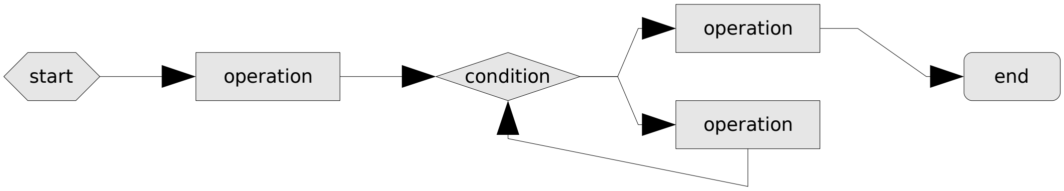
All programming languages use their own paradigm — or metaphor — to represent in a simple to understand yet very usable manner what the underlying hardware of the computer really does. Each language has its own paradigm: they all deal with data, program flow, user interaction, computer interaction and so on in somewhat different ways; there nonetheless are some basic features that most programming languages (like Fortran and C) use. Before learning a language perse we will learn about these few basic paradigms to get a good idea of what a computer can do.

Why are there multiple languages and multiple paradigms used? Some languages are better adapted to certain types of problem solving: they might be easier to use in certain situations or might be more efficient in the way the paradigm used translates into machine code; other languages might have bigger libraries available to them; others might be more in use; others might present some interesting features to the programmer while making some trade offs; and other languages might be more suited to a particular deployment platform.

Program Flow

- A program always starts running form the same entry point
- It then branches off and follows:
  - Loops
  - Conditions
- Tree-like structure that is well represented with a flowchart
- Branching depends on: user interaction, data values, interaction with other machines, …

The following flowchart illustrates these basic concepts of operations, conditions, branching and looping.



Programming concepts

- Language keywords, syntax and formatting
- Variables
- Data types
- Joining data together in structures such as arrays; concept of indexing
- Operations and operators (unary, binary and ternary)
- Conditions
- Looping (doing the same thing over and over again until a condition is met)
- Initial conditions
- Exit conditional
- Increment
- User interaction
- Functions:
    - Reuse code
    - Breaks the problem down into elemental pieces (divide and conquer)
    - Easier to test
    - Allows one to use bits of code as a “black box” knowing only its interface and not the specifics of the implementation (know only what is done and not how it is done)
- Discretization of data: transforming continuous data into discrete points, for example in stress analysis or in digital signal sampling
- Software writing principles
- Testing often, and using the errors and warnings given by the compiler
- Meaningful variable names (self documenting code)
- Software commenting:
    - Comments are ignored totally by the compiler
    - Comments allows the programmer to explain what a part of the code is doing to other programmers reading his code or for his own future reference
    - “Real programmers don't comment: it was hard to write, it should be hard to read!” — false!
- Software formatting (using whitespace properly when allowed in the language)
- Possible program errors
    - Does not compile
        - Syntax error
        - Logic error
        - Programming feature usage error (function, data types, etc…)
    - Compiles but crashes at runtime
        - Variable overflow, or reaching other software implementation limit
        - Illegal memory access, or other illegal resource access
        - Unhandled exception
    - Compiles, runs without crashing, produces unexpected results
        - Logic error
        - Rounding problems
        - Type conversion causing loss of data
    - Compiles, runs without crashing, produces results that look good but aren't: this is the most dangerous type of error, and is common at an advanced level

Command line interface basics

Under Windows

- Navigation with `cd`, `cd..`
- Directory listing with `dir`
- Running a program by being in the same folder and typing its name (`.exe` extension is optional)
- Running a program by typing its absolute path
- Passing arguments to a program
- Input and output redirection with < and >

Under Mac OS X

- Navigation with `cd`, `cd ..`
- Directory listing with `ls`
- Running a program by being in the same folder and typing its name (`.exe` extension is optional), note that a `./` before the program name is required to run a program in the current directory
- Running a program by typing its absolute path
- Passing arguments to a program
- Input and output redirection with < and >

Compiling

- From the command line interface, Fortran code is compiled with:
  ```
  g77 -x f77 -ffree-form -W -Wall "filename.f90" -o "filename.exe"
  ```
- From the command line interface, C code is compiled with:
  ```
  gcc -W -Wall "filename.c" -o "filename.exe"
  ```
- If you use SciTE as your text editor (provided on WebCT), these can be automated (more on saving some typing will be shown in tutorials)
- Handling compiler errors and warnings, what the difference is between an error and a warning, and why they shouldn't be ignored

General computer terminology

- RAM, ROM
- CPU, GPU
- Clock rate: Hz, MHz, GHz
- Bits and bytes (KB, MB, GB, TB)

III Starting to Program in Fortran

Introduction

Fortran will be the first programming language taught in this class. The programs we will be writing will be relatively trivial and not very engineering oriented, they will nonetheless allow us to use all the building blocks previously seen and who knows, they might even be fun!

Some Fortran specific facts

- Not whitespace-insensitive
- case-insensitive (make a clear distinction between data and source code case insensitivity), in our examples Fortran keywords are in uppercase and everything else is in lowercase
- The concept of “blocks”, and using whitespace to delimit those blocks

Basic Fortran programming

- The `PROGRAM` block:

```
PROGRAM program_name
  IMPLICIT NONE
  variable_declarations

  statements

END PROGRAM
```

- `WRITE(*,*)` statement used to write something to the screen, as in:

```
PROGRAM hello
  IMPLICIT NONE
  ! This is my first program.

  WRITE(*,*) "Hello World!"

END PROGRAM
```

- Comments start at an `!` and go until the end of the line
- Variable declaration must be at the beginning of the `PROGRAM` block before any executable statement:

```
PROGRAM program_name
  IMPLICIT NONE
  INTEGER variable_name1, variable_name2
  REAL variable_name3, variable_name4

  statements

END PROGRAM
```

  Variables can have any name, as long as it's an alphanumeric combination that starts with a letter, variable names should not be the same as Fortran keywords

- Two basic data types: `INTEGER` and `REAL`
- Using `WRITE(*,*)` to write the value of one or multiple variables to the screen
- Basic arithmetics: as in Mathematics, the following operations are available though with some limits in precision and some restrictions in integer division

+	-	*	/	**
addition	subtraction	multiplication	division	exponentiation

- Assignment of a value to a variable with `=` where the variable is on the left and the value on the right, **never** the other way around!
- Operator precedence and associativity
- Forcing a certain order of evaluation with parenthesis
- Arithmetic precision and limitations on values contained within variables
- Mixed arithmetics (and a very important note on integer division)

IV More Complicated Fortran Programming

More Fortran features

- Using `READ(*,*)` and `WRITE(*,*)` for user interaction
- `CHARACTER` variables (and declaring their length with `CHARACTER LEN=`*`number`*)
- Arrays and their indexing

Loops

Loops allow a program to do the same thing over and over again, which is very useful with arrays.

- `DO` loops
- Infinite `DO` loops
- Inlined `DO` loops
- `DO WHILE` loops
- `CONTINUE` and `EXIT` statements

Conditionals

- `LOGICAL` variables: either `.TRUE.` or `.FALSE.` (notice the dots)
- `IF THEN END IF` statement
- `ELSE IF` and `ELSE` statements
- One line `IF` statement
- Comparison operators
  - == to test for equality (watch out, a single equal sign is for assignment!)
  - /= to test for inequality
  - < less than
  - <= less than or equal
  - > greater than
  - >= greater than or equal
- Logical operators are used to group multiple comparisons together
  - `&&` stands for “and”
  - `||` stands for “or”
- To switch a comparison from `.TRUE.` to `.FALSE.` or from `.FALSE.` to `.TRUE.` use the unary `.NOT.` operator

Built-in functions

The `MOD(a, b)` function returns the remainder of the division of `a` by `b`

V Advanced Fortran Features

Formatting input and output

- `format_label FORMAT( format_list )`
- Different formats, where *n* is the number of items to be processed, *w* is the total width occupied by one item, and *d* is the number of places after the decimal point. In all cases these numbers are optional, if omitted standard values will be used.

`nIw`	*n* integers each occupying a width of *w*
`nFw.d`	*n* floating point numbers, each occupying a width *w* of which *d* are after the decimal point (do not forget the point itself occupies a space)
`nEw.d`	Similar to above, but in exponential form
`nGw.d`	Similar to above, real or integer (general form)
`nLw`	*n* logicals each occupying a width of *w*
`nAw`	*n* character variables each occupying a width of *w*
`'text'`	Outputs what is between quotes as is
`r(format_list)`	Repeats the format in round braces *r* times

There are other formats not covered here since text processing isn't a main objextive of this engineering class.

- `WRITE(*,format_label)` and `READ(*,format_label)` using a `*` as format label means using the default format
- Formatting using `CHARACTER` variables or constants

File interaction

- `OPEN(UNIT = unit_number, FILE = "file_name")`
- Note that unit numbers and format numbers are in distinct name spaces, moreover if a format is used to read or write to a file the numbers used need not be the same
- `READ(unit_number,*)` and `WRITE(unit_number,*)` to/from the opened file using a `*` as format number means using the default “file”: the command line interface
- `CLOSE(unit_number)`, and why it's a Good Thing to close files that aren't needed anymore

Functions and subroutines

Introduction

As seem before, functions and subroutines are used to:

- Reuse code
- Breaks the problem down into elemental pieces (divide and conquer)
- Easier to test
- Allows one to use bits of code as a “black box” knowing only its interface and not the specifics of the implementation (know only what is done and not how it is done)

Syntax

The FUNCTION and SUBROUTINE blocks are similar to the PROGRAM block, and they should be inserted outside of the PROGRAM block:

```
PROGRAM program_name
  IMPLICIT NONE
  variable declarations

  program body

END PROGRAM


SUBROUTINE subroutine_name(comma separated parameter list)
  IMPLICIT NONE
  variable declarations

  subroutine body

END SUBROUTINE


return_type FUNCTION function_name(comma separated parameter list)
  IMPLICIT NONE
  variable declarations

  function body

END FUNCTION
```

Usage

- Using a function, and returning a value from it
- Using the `CALL` keyword to call a subroutine
- Variable scope
- Passing arguments (by reference always): passing arguments to a function or subroutine allow that function or subroutine to access the passed variables from the calling `PROGRAM`, `FUNCTION` or `SUBROUTINE` block.
- Passing arrays to a function or subroutine

VI Transition from Fortran to C

Introduction

The second language taught this class is C. Today's lecture will be a crash course on C, showing how to translate the basic concepts seen from Fortran to C. Contrarily to what was done in Fortran, we will be using C mostly to solve typical engineering problems, that is we will be showing C while we also show algorithms.
Also, in Fortran we didn't really care about how programming features were implemented and how the computer itself really worked, in C we will outline this more to allow us to write more efficient code: our programs will be aimed at speed of execution and space saving.

Hello World

```c
#include <stdio.h>

int main()
{
  printf("Hello World!\n")

  return 0;
}
```

This program does the following:

- Tell the compiler to include functions, data types and others from the standard input/output library (`stdio.h`) which is one of the standard libraries that come with the C language but that are not part of the language perse. The C language does not include any built-in functions into a program until header files are called by the programmer, asking the compiler to include these functions.
- Start the `main` function, whose return type is `int` (for integer), this is the entry point of our program.
- Print to the screen (the command prompt) the characters `Hello World!` And put a newline after this — the `\n` denotes this newline — `printf` is a function found in the standard input/output library.
- Return `0` to the calling program, signifying that the program terminated correctly (anything other than `0` means otherwise), this is an integer as was expected from `main`'s declaration.

C programs are compiled much like Fortran programs are, as shown in II Basic Computer Concepts.

Quick comparison table

Concept	***Fortran***	***C***
Entry point	PROGRAM block	main function
Control blocks	FUNCTION and SUBROUTINE Call by reference is default	Functions Call by value is default
Code formatting	Whitespace sensitive	Whitespace insensitive
	One statement per line	Statements **must** end with a semicolon and can be on the same line
	Case insensitive (save for data)	Case sensitive
Comments	Single line: ! to end of line	Single line: // to end of line Multi line: /* to */
Data types	INTEGER	int
	REAL	float or double
	CHARACTER	char for a single character char[] for a character string
	LOGICAL	int
	No equivalent	void
	POINTER (not seen in this class)	Pointers, unary * and & operators
Arrays	Column major	Row major
	1D array: *arr_name*(*size*)	1D array: *arr_name*[*size*]
	2D array: *arr_name*(*size_1*, *size_2*)	2D array: *arr_name*[*size_1*][*size_2*]
	First index is 1, last is size	First index is 0, last is size-1

Interaction	READ(*src*,*fmt*) *var_1*, *var_2*
	WRITE(*dst*,*fmt*) *var_1*, *var_2*
		scanf(*fmt*, &*var_1*, &*var_2*);
		printf(*fmt*, *var_1*, *var_2*);

Conditions	IF (*condition*) THEN
	  statements
	ELSE IF(*condition*) THEN
	  statements
	ELSE
	  statements
	END IF
		if(*condition*){
		  statements
		} else if(*condition*){
		  statements
		} else {
		  statements
		}
		Curly braces can be omitted if there is only one statement
		Do **not** put a ; after the closing)

Logical comparison operators	==	==
	/=	!=
	<	<
	<=	<=
	>	>
	>=	>=
	&&	&&
	||	||
	.NOT.	!

Concept	Fortran	C

Loops	DO var = val, end_val, increment
	  statements
	END DO
		for(
		  init_statement;
		  exit_statement;
		  increment_statement
		){
		  statements
		}
		
		or
		
		init_statement;
		while(exit_statement){
		  statements
		  increment_statements;
		}
		
		Note that these for and while loops are not strictly the same but are close enough that in this class you probably won't notice the difference
		Curly braces can be omitted if there is only one statement
		Do **not** put a ; after the closing)

	DO
	  statements
	END DO
		for(; ;){
		  statements
		}

	Inlined DO loop	No equivalent, but using printf does not append newlines like WRITE does

	DO
	  statements
	WHILE(exit_statement)
		do{
		  statements
		}while(exit_statement);

	CONTINUE	continue;
	EXIT	break;
Other operators and features	Arithmetic: + - * /	Arithmetic: + - * /
	MOD function	% operator
	Exponentiation: **	pow function, in math.h
	No equivalent seen	Post- and pre- incrementation and decrementation with the ++ and -- operators
	Assignment: =	Assignment: =
	No equivalent seen	Bitwise operators: & | ^ ~ << >> (not covered in details in this class)
	No equivalent seen	Ternary operator: ? :
	No equivalent seen	Condensed operators: += -= *= /= %= &= ^= |= <<= >>=
	No equivalent seen	Casting: (datatype)
	No equivalent seen	Pointer-related: & * ++ --

More details on C

List of various details

- The `main` function
- Declaring variables: as in Fortran, they must be declared at the beginning of any function before any executable statement
  Variables can have any name that is an alphanumeric combination, it can contain underscores and must starts with either a letter or an underscore, but that name should not the a C keyword
  Declaring variables in global scope, and why it should be avoided most of the time
- Core language data types:
  - Integral types:
    - `int` (for integer)
    - `char` (for character)
      - A `char` is a single character like the letter `'a'` or the digit `'4'`
      - `char` literals are enclosed in single quotes
      - Special characters can use an escape notation by being preceded by a backslash, of note is the `'\0'` null character
      - For more on this see appendix B
  - Floating point types:
    - `float`
    - `double` (more precise than `float`s, `double`s should be used instead of `float`s in most cases)
  - `void` type, and what it means for something to have `void` type
  - There is no logical or boolean type in C: instead `int`s are used and anything that is zero is false, everything that is non-zero is true
- Arrays:
  - Use square braces
  - Arrays of dimension higher than one are indexed as: `arr[2][3]`
  - Row major
  - First index is `0`, last index is `size-1`
  - Passing an array to a function (to be seen later)
- Character string:
  - An array of `char`s is a character string
  - Null terminated (the `'\0'` character)
  - Character string literals are enclosed in double quotes
  - Character string declaration, initialization and indexing
- Modifiers:
  - `short`
  - `long`
  - `long long`
  - `unsigned` (and `signed`)
  - `static`
  - `const`
- Usual size of variables and their precision:

- `char` 1 byte: 2^8 (256) possible characters
- `short int` 2 bytes: -2^{15} to $+2^{15}-1$ (-32,768 to +32,767)
- `int` 4 bytes: -2^{31} to $+2^{31}-1$ (roughly -2 billion to +2 billion)
- `long int` 4 bytes on 32 bit processors, 8 bytes on 64 bit processors
- `long long int` 8 bytes: -2^{63} to $+2^{63}-1$ (roughly -9 quintillion to +9 quintillion)
- Using `unsigned`
- `float` 4 bytes:
  - 1 bit sign
  - 8 bits exponent (-126 to +127)
  - 23 bits fraction (9 significant digits)
- `double` 8 bytes:
  - 1 bit sign
  - 11 bits exponent (-1,022 to +1,023)
  - 52 bits fraction (17 significant digits)

Functions

- Declaration
- Prototypes
- Passing arguments (by value always): contrary to Fortran, arguments passed to a C function are always passed by value, this means that the called function makes a copy of the value of the variables passed to it, thereby preventing it from modifying the value contained within the original variable.
- Returning a value (or not) from a function

Pointers

Pointers are declared as other variables are, but they are preceded by a star in their declaration:

```
  datatype normal_var1, *pointer_var, normal_var2;
```

A pointer is an integer variable itself, this integer represents the address of a variable in memory. Indeed a computer's memory is similar to a single very long street with houses (variables) along it, each having an (integer) address which allows us to find that house. A pointer therefore “points” to another variable, much like a piece of paper with the address of a friend's house allows us to find that friend.

In our example, the street is the computer's memory, the houses are the variables and each have an address, and the friend living in the house is the data contained withing the variable.

To obtain the address of a variable, we precede its name by the unary `&` operator, therefore doing `ptr_to_a = &a;` makes the variable `ptr_to_a` point to the variable `a` by having that pointer hold the address of the variable `a`.

Pointers are especially useful to allow a function to access variables defined within a function that we want to use in a function called by the original function: indeed parameters passed to a function in C are “passed by value”, meaning that the called function makes a copy of the data before acting on it. This means that the called function cannot return that data to the calling function except by using the return statement, which allows us to return a single variable (as far as we'll see in this class). Therefore passing to the called function the address of a variable allows the called function to know where the calling function's variable is in memory, and thus act directly upon it instead of acting upon a local copy of that data.

Pointers also come in handy when working with array: indeed it is guaranteed in C that arrays are stored continuously in memory, meaning that an element whose index is `N` is located in memory between elements `N-1` and `N+1` (provided these exist). Therefore knowing the address of the first element of an array (with a pointer) and knowing the total number of elements of an array allows us to know what the address of every single element of that array is.

This might all seem very complex for now, but as we start using pointers in the next classes it should become clearer. A few more things that we will learn to do with pointers are:

- The pointer/array duality
- Incrementing/decrementing a pointer
- Dereferencing a pointer
- Pointers to functions
- `typedef` to declare new types

Pointers also happen to have lots of other nice and fun uses, but we won't get to see all of them in this class.

Preprocessor

Preprocessor commands are lines starting with a `#`.

Preprocessor commands are evaluated before the program is compiled, they allow the source file to be modified before the compiler sees it, for example to include other files or to change some text. Preprocessor commands are never terminated by semicolons.

- Header files are included with:
  ```
  #include <filename>
  ```
- Our own header files can be included with the file name in double quotes:
  ```
  #inlude "myfile.h"
  ```

The following will probably not be used in this class but are provided here for completeness:

- Preprocessor variables (called symbolic name or symbolic constant) are usually written in uppercase to distinguish them from program variables and are defined as:
  ```
  #define NAME value
  ```
  For example to replace every occurrence of `ANSWER` in a source file with `42`:
  ```
  #define ANSWER 42
  ```
- Preprocessor functions are defined as:
  ```
  #define FUNC_NAME(paramaters) func_body
  ```
  For example a function that, when called, replaces a variable by a variable times itself (note the use of parenthesis to prevent any precedence problems):
  ```
  #define SQUARE(x) (x*x)
  ```
- Preprocessor conditional statements
  - `#ifdef`
  - `#ifndef`
  - `#else`
  - `#endif`

VII Some More of C, Using Standard Libraries

stdio.h

- `printf("`*format*`", `*arg1*`, `*arg2*`, …)`
  - `%d` or `%i` for integers
  - `%f`, `%e` or `%g` for floating point numbers (respectively floating point, exponential or shortest possible notation)
  - `%c` for characters
  - `%s` for strings (must be null terminated)
  - Insertion of newline characters `\n`
  - The size of the field can be set in advance by including it between the `%` and the letter, for example to force an integer to take four spaces we can do `%4i`
  - The `f` conversion can also specify how many digits of precision to put after the dot, to put five we can do `%.5f` (note the dot)
    both total field size and precision can be present, for example `%10.5f`
  - Other conversions and options exist, but the are not covered in this class
- `scanf("`*format*`", &`*arg1*`, &`*arg2*`, …)`
  - Note that a pointer needs to be passed to this function
  - Reading into a string
  - Reading into an array
  - Formats are similar to those for `printf`, but to read doubles use `%lf`
  - For this class, formats should not contain any other characters than the conversion characters and whitespace
- `FILE*` datatype
- `FILE *fopen("`*filename*`", "`*mode*`")`
  Where the mode is `w` or `r` (others exist but are not discussed in this class)
- `fclose(`*FILE_variable*`)`
- `fprintf(`*FILE_variable*`, "`*format*`", `*arg1*`, `*arg2*`, …)`
- `fscanf(`*FILE_variable*`, "`*format*`", &`*arg1*`, &`*arg2*`, …)`

math.h

All basic mathematical functions and a few mathematical constants, note that angles are provided in radians.

time.h

- `time(NULL)` returns the number of seconds elapsed since an arbitrary time
  Its return type is of data type `time_t`
- `clock()`, `clock_t` data type and `CLOCKS_PER_SEC` preprocessor variable

Others to be seen later

limits.h

Contains information on the limits of the current implementation.

stdlib.h

Contains a few useful functions such as:
`calloc`, `malloc`, `free`, `exit`, `srand`, `rand`, `abs`

string.h

Contains functions related to string handling.

VIII Searching

What is an algorithm?

An algorithm is a procedure that is followed to solve a particular problem given certain initial conditions There always are multiple methods for solving the same problem, therefore some algorithms are more efficient than others time-wise, storage-wise or in some other manner.

The following lectures will show a few basic algorithms used in engineering and computer sciences to help the students learn to design algorithms, and to optimize their code. All the algorithms show are available on WebCT in the `algorithms.zip` file.

Common errors

In programs like these programming errors often happen when dividing even or odd lists, or when the data set reaches a small size. Oftentimes errors are off-by-one types of errors, or errors that happen in “that very rare case” in which an algorithm can end up, therefore we must be very sure that we think about every possible outcome of our program.

Big-O

Measure the order of time an algorithm takes to execute.
Examples assume `i`, `j` and `N` are integers, and `A` is and integer array of size `N`.

O(N)

```
for(i = 0; i < N; ++i) {
  A[i] = i;
}
```

O(log N)

```
for(i = 2; i < N; i *= 2) {
  A[i] = i;
}
for(i = N; i > 0; i /= 2) {
  A[i] = i;
}
```

O($N^{1/2}$)

```
for(i = 2; i < N; i *= i) {
  A[i] = i;
}
```

O(N log N)

```
for(i = 0; i < N; ++i)
  for(j = 2; j < N; j *= 2) {
    A[i] = i+j;
  }
```

O(N^2)

```
for(i = 0; i < N; ++i)
  for(j = 0; j < N; ++j) {
    A[i] = i+j;
  }
```

Pointer introduction

Pointers introduction, passing an array to a function, array/pointer duality.

Linear search

- Search from the first element in the list onwards, until the element searched for is found or the end of the list is reached
- In average, one can expect to find a solution after ½ N iterations, it is therefore said that linear search is, close to a constant k equal to ½ here, “of order N” or O(N)
- Works on unordered or ordered lists

```c
int linear_search(int val, int arr[], int size)
{
  int i;

  // Go through each item one by one, looking for the value.
  for(i = 0; i < size; ++i) {
    if(arr[i] == val)
      return i;
  }

  // Value not found, return an impossible index.
  return -1;
}
```

Binary search

- List **must** be ordered, here we'll assume it is non-decreasing
- Knowing that the list is ordered, use the intermediate value theorem: if the value we're looking for is smaller than the number at the middle of the list then it must be in the lower half of the list, if it's greater than the number at the middle of the list then it must be in the other half. Apply the same principle, dividing the list in two over and over again until the value searched for is found or the search turns up nothing (meaning that the value isn't in the list).
- O(log N)

```
int binary_search(int val, int arr[], int size)
{
  int left = 0, right = size, middle;

  do {
    middle = (left + right) / 2;

    if(arr[middle] < val)
      left = middle + 1;
    else if(arr[middle] > val)
      right = middle - 1;
    else
      return middle;

  } while(left <= right);

  // Value not found, return an impossible index.
  return -1;
}
```

Finding biggest and smallest

Similar to linear search.

Find biggest

```c
int find_biggest(int arr[], int size)
{
  int index_of_big = 0, i;

  for(i = 0; i < size; ++i)
    if(arr[i] > arr[index_of_big])
      index_of_big = i;

  return index_of_big;
}
```

Find smallest

```c
int find_smallest(int arr[], int size)
{
  int index_of_small = 0, i;

  for(i = 0; i < size; ++i)
    if(arr[i] < arr[index_of_small])
      index_of_small = i;

  return index_of_small;
}
```

IX Sorting

Introduction

All the sorting methods to be seen today are of order O(N^2) and aren't the best methods around; they nonetheless allow us to practice writing fairly easy algorithms.

New features

- Learn about pointers again
- Swapping, and pass-by-value versus pass-by-reference

Bubble sort

Lighter elements “bubble” to top of list: go through the list comparing each adjacent elements, if they're not in order swap them. Doing one pass of this guarantees that the smallest element will end up at the beginning of the list. Go through the list again until all the elements are in order, a total of N passes (N being the total number of elements in the list) need to be made for all the list to be in order.
A few optimizations can be made:

- If we haven't swapped in a pass then the list is in order, therefore we can stop
- We know that after X passes the X smaller elements are at the beginning of the list, it therefore isn't useful to pass through these elements again, therefore pass X will have NX comparisons instead of N

```
void bubble_sort(int arr[], int size)
{
  int i, j, swapped;

  for(i = 0; i < size - 1; ++i) {
    swapped = 0;

    for(j = size - 1; j > i; --j)
      if(arr[j] < arr[j - 1]) {
        swap(&arr[j], &arr[j - 1]);
        swapped = 1;
      }

    if(!swapped)
      break;
  }

  return;
}
```

Selection sort

Go through the list looking for the smallest element and place it at the beginning of the list. Do the same thing in the remaining list, a total of N times.

```c
void select_sort(int arr[], int size)
{
  int i, index_of_small;

  for(i = 0; i < size; ++i) {
    index_of_small = find_smallest(arr + i, size - i);
    swap(arr + i, arr + i + index_of_small);
  }

  return;
}
```

Insertion sort

Start with a list of 1 element, and grow it progressively to N elements by adding a new element on each pass at the end of the list and bubbling it to its proper place.

```c
void insertion_sort(int arr[], int size)
{
  int i, j;

  for(i = 1; i < size; ++i)
    for(j = i; j; --j)
      if(arr[j] < arr[j - 1])
        swap(&arr[j], &arr[j - 1]);
      else
        break;

  return;
}
```

X Recursion

Introduction

Recursion is the process that allows the “divide and conquer” idea to take place: the problem is divided into smaller and smaller bits until the problem is in an easily solvable form, meaning that the same algorithm is applied to subsets of the same data over and over again until the subset of the data is a subset for which the problem is easy to solve.

Everything that is expressed as a loop can be expressed recursively, solving a problem with a loop usually follows these steps:

1. Set up the data (not always present)
2. Do the same operation on the data over and over again until a certain condition is met
3. Do a final operation on the data (not always present)

Recursion is a very different way of seeing a problem, and it does take quite some time and a few examples to understand, nonetheless it is a very powerful concept that allows complex problems to be solved in a very simple manner. While recursive functions might at first sight seem circular and never ending, one must always keep in mind that the most important part of a recursive function is/are the termination condition(s); one of these conditions must be reached in each “branch” of recursion if the function is to ever end.

It is often said that “to understand recursion, one must first understand recursion”.

Simple recursion examples

Factorial, recursively

```c
unsigned int factorial(unsigned int n)
{
  if(n)
    return n * factorial(n - 1);
  else
    return 1;
}
```

Factorial, non-recursively

```c
unsigned int factorial_nonrec(unsigned int n)
{
  unsigned int result = 1, i;

  for(i = 1; i <= n; ++i)
    result *= i;

  return i;
}
```

McNugget numbers

A number is said to be “McNugget” if it can be obtained by adding together orders of McDonald's Chicken McNuggets, which originally came in boxes of 6, 9, and 20.

```
int is_mc_nugget(int n)
{
  return
    ((n >= 20) && is_mc_nugget(n - 20)) ||
    ((n >=  9) && is_mc_nugget(n -  9)) ||
    ((n >=  6) && is_mc_nugget(n -  6)) ||
    (n == 20) || (n == 9) || (n == 6);
}
```

To understand this function, first you need to notice that all it really is is a complex combination of logical operators: the logical or `||` and the logical and `&&`. This entire expression (what's after the return) will be evaluated and that's what will be returned, and because it's a huge logical operator combination it will evaluate to true or false (non-zero or zero). Let's break it down more:

```
int is_mc_nugget(int n)
{
  return // This means: return what's after this, until the semicolon.
  // Now notice something: all of what follows if something OR something else
  // OR something else OR... Therefore this entire thing will be true if any one of these OR is true.
  // If our number is greater than or equal to twenty, and that that number minus 20
  // is McNugget itself, then this number is McNugget.
    ((n >= 20) && is_mc_nugget(n - 20)) ||
  // If our number is greater than or equal to nine, and that that number minus 9
  // is McNugget itself, then this number is McNugget.
    ((n >=  9) && is_mc_nugget(n -  9)) ||
  // If our number is greater than or equal to six, and that that number minus 6
  // is McNugget itself, then this number is McNugget.
    ((n >=  6) && is_mc_nugget(n -  6)) ||
    (n == 20)  || // If this number is twenty, then it's McNugget.
    (n == 9)  || // If this number is nine, then it's McNugget.
    (n == 6)  // If this number is six, then it's McNugget.
    ;  // End that huge return statement.
}
```

You now have to convince yourself that this recursiveness will end one day: you should agree that eventually in the recursive chain our number n will be either smaller than 6 or equal to 6, 9 or 20, you should then agree that if it gets smaller than 6 that means that the statement will return false, and if it's equal to 6, 9 or 20 it'll return true.

Now for the recursive part: we're saying “if n is greater than or equal to 20 and n minus 20 happens to be McNugget, then n itself must be McNugget”, this is fairly obvious: it's like buying a box of 20 McNuggets. The same applies individually for 9 and 6. But how does it all come together? What's really happening is that you try all combinations possible: you try removing, in all possible quantities, 20, 9, and 6 from your original number and you try to see if that ends up giving you a McNugget number with only one box (6, 9 or 20), or if you end up with a number smaller than 6 (in which case your number is non-McNugget).

Rewriting searching and sorting algorithms recursively

Binary search

```
int recursive_binary_search(int val, int arr[], int size)
{
  int i = size / 2;

  if(size == 1) {
    if(arr[0] == val)
      return 0;
    else
      // Value not found, return an impossible index.
      return INT_MIN;
  }

  if(arr[i] > val)
    return recursive_binary_search(val, arr, i);
  else if(arr[i] < val)
    return i + recursive_binary_search(val, arr + i, size - i);
  else if(arr[i] == val)
    return i;
}
```

Bubble sort

```
void recursive_bubble_sort(int arr[], int size)
{
  int i;

  if(size <= 1)
    return;

  for(i = size - 1; i; --i)
    if(arr[i] < arr[i - 1])
      swap(&arr[i], &arr[i - 1]);

  recursive_bubble_sort(arr + 1, size - 1);

  return;
}
```

Selection sort

```
void recursive_select_sort(int arr[], int size)
{
  int index_of_small;

  if(size <= 1)
    return;

  index_of_small = find_smallest(arr, size);
  swap(arr, arr + index_of_small);

  recursive_select_sort(arr + 1, size - 1);

  return;
}
```

Insertion sort

```
void recursive_insertion_sort(int arr[], int size)
{
  int i;

  if(size <= 1)
    return;

  recursive_insertion_sort(arr, size - 1);

  for(i = size - 1; i; --i)
    if(arr[i] < arr[i - 1])
      swap(&arr[i], &arr[i - 1]);
    else
      break;

  return;
}
```

XI Merge Sort

Introduction

The easiest list to sort is a list with only one or two items: with one item the list is always sorted, and with two items either the list is sorted or it isn't, in which case swapping the two elements makes the list sorted. That is the driving idea behind merge sort. This seems very trivial but combined with recursion (which we saw in the last class) and a clever “merging” idea, this trivial idea allows us to program one of the most efficient sorting algorithms around. Indeed, recursion is used until the data set is in a manageable form of either one or two elements, these are sorted and merged back with the other data sets to form a sorted list.

The “merging” bit is quite simple: two lists that we know are sorted are to be joined together in a single list, we know that the smallest item from the final list is either the first item of the first list, or the first item of the second list (assume here that this was the case). Then the second biggest item of the final list has to be (in our example) the first item of the first list, or the second item of the second list. This is applied on and on until both lists are empty, therefore until both lists have been merged into a single ordered list.

Note that merge sort requires a temporary array that has a size equal to the original array.

This algorithm is of order O(N log N).

Interesting demos of sorting: http://cg.scs.carleton.ca/~morin/misc/sortalg/

New features

New language features and standard library elements used:

- `malloc`
- `free` and why it's a Good Thing to `free` memory that isn't needed anymore
- `sizeof` keyword
- `stdlib.h`
- `memory.h`

Code

```
static void _merge(int left[], int left_size,
  int right[], int right_size, int destination[])
{

  int left_i = 0, right_i = 0, destination_i = 0;

  // Choose the smallest element from the left or right array, and put it into the destination.
  // Do this until the end of one of the arrays is reached.
  while((left_i < left_size) && (right_i < right_size))
    if(left[left_i] < right[right_i])
      destination[destination_i++] = left[left_i++];
    else
      destination[destination_i++] = right[right_i++];

  // Put any remaining elements from the left or right arrays into the destination array.
  while(left_i < left_size)
    destination[destination_i++] = left[left_i++];
  while(right_i < right_size)
    destination[destination_i++] = right[right_i++];

  return;
}


static void _merge_sort(int arr[], int size, int temporary[])
{
  int half = size / 2;

  if(size <= 1)
    return;

  // Recursively sort left half.
  _merge_sort(arr, half, temporary);
  // Recursively sort right half.
  _merge_sort(arr + half, size - half, temporary + half);

  // Merge them back together.
  _merge(arr, half, arr + half, size - half, temporary);

  // Copy size elements from temporary into arr.
  memcpy(arr, temporary, size * sizeof (int));

  return;
}

```

```
// Initialize merge sort, this is the function that you should be calling.
void merge_sort(int arr[], int size)
{
  // Allocate the temporary array.
  int *temporary = (int *) malloc(size * sizeof (int));

  // Start the recursive sort.
  _merge_sort(arr, size, temporary);

  // Free the allocated array.
  free(temporary);

  return;
}
```

XII Root Finding, Numerical Differentiation

Introduction

Root finding allows us to find where a certain function is equal to zero. We will only deal here with functions that take a single argument, and we will assume that the functions are continuous and differentiable on the interval studied.

New features

- Pointers to function, passing a function to another function
- typedef

```
typedef double (*DfD) (double);
typedef double (*DfDD) (double, double);
typedef double (*DfDDD) (double, double, double);
```

Bisection

- Function must be continuous and have exactly one root in the interval, it might work or might fail otherwise.
- Much like binary search: divide the interval in two and look for a sign change in the first half of the interval provided. From the Intermediate Value Theorem if there is a sign change in the first half then the root must be in the first half, or in the other half if there isn't a sign change.
- Note that we are using doubles, we therefore must provide a tolerance to how close we want our result to be to zero if we do not want to rely on the implementation's default value (which might be very small and might make our program run for ever).

```
double bisection_rf(DfD f, double x0, double x1, double tol)
{
  double middle = (x0 + x1) / 2.0;

  if((middle - x0) < tol)
    return middle;
  // From the Intermediate Value Theorem, if there is a sign change then there is a root.
  else if(f(middle) * f(x0) < 0.0)
    return bisection_rf(f, x0, middle, tol);
  else
    return bisection_rf(f, middle, x1, tol);
}
```

Secant

- The two initial guesses do not have to bracket the root
- Select two points on the function and draw a line, getting closer to a root, use this guess as the next point
- Note that since we are not bracketing the root there is a possibility that the function will never stabilize (as it might on the `sin` function, for example), we therefore provide a maximum number of steps that the function is allowed to take.

```
double secant_rf(DfD f, double x1, double x2, double tol, int count)
{
  double f1 = f(x1), f2 = f(x2),
    slope = (f2 - f1) / (x2 - x1),
    distance = -f2 / slope, point = x2 + distance;

  if(!count)
    return point;

  if(fabs(distance) < tol)
    return point;

  return secant_rf(f, x2, point, tol, count - 1);
}
```

False-Position

- Bracket root, and combine bisection with secant

```
double fixedpoint_rf(DfD f, double x1, double x2, double tol)
{
  double f1 = f(x1), f2 = f(x2),
    slope = (f2 - f1) / (x2 - x1), distance = -f1 / slope,
    point = x1 + distance;

  if(((point - x1) < tol) || ((x2 - point) < tol))
    return point;

  if((f1 * f(point)) < 0)
    return fixedpoint_rf(f, x1, point, tol);
  else
    return fixedpoint_rf(f, point, x2, tol);
}
```

Newton-Raphson

- Much like secant, but using the real derivative: mathematically using two initial guesses that are infinitely close

```c
double newton_rf(DfD f, DfD df, double x, double tol, int count)
{
  double distance = -f(x) / df(x);

  if((fabs(distance) < tol) || !count)
    return x + distance;

  return newton_rf(f, df, x + distance, tol, count - 1);
}
```

Newton-Raphson using numerical differentiation

```c
double newton2_rf(DfD f, double x, double tol, int count)
{
  double distance = -f(x) / centered3_diff(f, x, 1e-6);

  if((fabs(distance) < tol) || !count)
    return x + distance;

  return newton2_rf(f, x + distance, tol, count - 1);
}
```

Numerical differentiation

Introduction

The mathematical definition of differentiation is: $f'(x_0)=\lim_{h\to 0}\frac{f(x_0+h)-f(x_0)}{h}$

This could be applied directly on a computer with a small value of h, but this will produce imprecise results if h is too big (the error is of order h), or roundoff errors if h is small. This imprecise method is called a “two point differentiation” formula since it uses two points on the f curve to approximate the derivative of f. Such two point formulas are very imprecise as explained above, we will therefore use the following methods instead.

We do not show how these formulas are derived, this is left to math classes.

Forward three point differentiation

Using three evenly spaced points forward of x, we get the formula:

```
double forward3_diff(DfD f, double x, double h)
{
  return (-3 * f(x) + 4 * f(x + h) - f(x + 2 * h)) / (2 * h);
}
```

This formula has an error of order h^2, thus although it is still subject to roundoff errors it still is accurate for relatively large values of h.

Backward three point differentiation

Similar to the above formula, but taking three points before x:

```
double backward3_diff(DfD f, double x, double h)
{
  return (f(x - 2 * h) - 4 * f(x - h) + 3 * f(x)) / (2 * h);
}
```

Centered three point differentiation

The following formula is similar to the above two, but takes a point before and a point after x. One of the function evaluations happens to cancel out, therefore this formula is to be used over the other two: it only has two function evaluations and has the same error in h^2.

```
double centered3_diff(DfD f, double x, double h)
{
  return (-f(x - h) + f(x + h)) / (2 * h);
}
```

Centered three point 2nd differentiation

This formula is included here solely for reference.

```
double centered3_2diff(DfD f, double x, double h)
{
  return (f(x - h) - 2 * f(x) + f(x + h)) / (h * h);
}
```

XIII Numerical Integration

Introduction

Mathematically, $\int_a^b f(x)dx$ is equal to the area under the curve of `f` from `a` to `b`. We will show three methods to calculate this, in all of them we assume that the function is continuously differentiable in the interval considered.
Since these methods do not use division the are less prone to severe rounding errors as differentiation was.

Midpoint

This technique consists of the approximation: $\int_a^b f(x)dx = \sum_{i=0}^{n} \frac{b-a}{n} f(x_i)$

This means that we're diving the interval in `n` rectangles and summing each rectangle's area. To be more accurate in our result, we take the point of evaluation of `f` at the middle of our interval. This gives us an error of order `h`^{2}, and from the nature of the formula this is 100% accurate on constant and linear functions.

```
double midpoint_int(DfD f, double x0, double x1, int n)
{
  int i;
  double sum = 0.0, x, dx = (x1 - x0) / n;

  // Offset initial point by dx/2 to be at the midpoint. Add the area of each panel,
  // assuming unit width and multiplying by the true width dx
  // only at the end to reduce rounding errors.
  for(i = 0, x = x0 + dx / 2.0; i < n; ++i, x += dx)
    sum += f(x);

  return sum * dx;
}
```

Trapezoidal

This method is similar to the previous and has the same order of error, but instead of adding up rectangles we add up trapezes which in average should give a better approximation.

```
double trapezoidal_int(DfD f, double x0, double x1, int n)
{
  int i;
  // Sum the outer edges divided by two.
  double sum = (f(x0) + f(x1)) / 2.0, x, dx = (x1 - x0) / n;

  for(i = 0, x = x0 + dx; i < n - 1; ++i, x += dx)
    sum += f(x);

  return sum * dx;
}
```

Simpson's

Instead of using a rectangle or trapeze, Simpson's rule fits a parabola through three points. This gives an error of order h^5 and a formula that is 100% accurate for polynomials of degree three or less.

```
double simpsons_int(DfD f, double x0, double x1, int n)
{
  int i;
  double sum = f(x1) - f(x0), x, dx = (x1 - x0) / n;

  for(i = 0, x = x0; i < n; ++i, x += dx)
    sum += 2.0 * f(x) + 4.0 * f(x + dx / 2.0);

  return sum * dx / 6.0;
}
```

Monte Carlo

Introduction

Monte Carlo integration is shown here as a concept only since it isn't very practical or accurate on functions taking only one parameter. It nonetheless is used on higher order functions in certain scientific fields, especially when the function is unstable. Monte Carlo simulation in general is used a lot to simulate something with random input.
The idea behind Monte Carlo integration is that we take random points in the interval, evaluate `f` there, and average the sum of these evaluations. In general as the number of sample points gets big this should give us a good approximation of the real integral.

New features

- `rand`, `srand`, `RAND_MAX`, how they work (pseudo-random number generator)
- seeding with time as an easy (but not robust) way to seed the pseudo-random number generator

Code

```
double monte_carlo_int(DfD f, double x0, double x1, int n)
{
  double sum = 0;
  int i;

  // Sum n random values of f(x) with x in [x0, x1].
  for(i = 0; i < n; ++i)
    sum += f(((double) rand() / RAND_MAX) * (x1 - x0) + x0);

  // Divide the sum by n to get the average value of f(x) with x in [x0, x1].
  // Multiply by (x1 - x0) to get the area.
  return (sum / n) * (x1 - x0);
}
```

XIV Initial Value Problem

Introduction

- Cellular automaton
- Marching algorithm idea

Forward Euler

$y(t+h)=y(t)+h\,f(t,y(t))$

In one variable

```
void euler1_step(double t, double *x, DfDD xp, double h)
{
  *x += h * xp(t, *x);

  return;
}
```

In two variables

```
void euler2_step(double t, double *x, double *y,
  DfDDD xp, DfDDD yp, double h)
{
  double x_cpy = *x;

  *x += h * xp(t, *x, *y);
  *y += h * yp(t, x_cpy, *y);

  return;
}
```

Runge-Kutta 4

In one variable

```
void runge_kutta1_step(double t, double *x, DfDD xp, double h)
{
  double h_half = h / 2.0, k1, k2, k3, k4;

  k1 = xp(t, *x);
  k2 = xp(t + h_half, *x + h_half * k1);
  k3 = xp(t + h_half, *x + h_half * k2);
  k4 = xp(t + h, *x + h * k3);

  *x += (h / 6.0) * (k1 + 2.0 * k2 + 2.0 * k3 + k4);

  return;
}
```

In two variables

```
void runge_kutta2_step(double t, double *x, double *y,
  DfDDD xp, DfDDD yp, double h)
{
  double h_half = h / 2.0, k1, j1, k2, j2, k3, j3, k4, j4;

  k1 = xp(t, *x, *y);
  j1 = yp(t, *x, *y);
  k2 = xp(t + h_half, *x + h_half * k1, *y + h_half * j1);
  j2 = yp(t + h_half, *x + h_half * k1, *y + h_half * j1);
  k3 = xp(t + h_half, *x + h_half * k2, *y + h_half * j2);
  j3 = yp(t + h_half, *x + h_half * k2, *y + h_half * j2);
  k4 = xp(t + h, *x + h * k3, *y + h * j3);
  j4 = yp(t + h, *x + h * k3, *y + h * j3);

  *x += (h / 6.0) * (k1 + 2.0 * k2 + 2.0 * k3 + k4);
  *y += (h / 6.0) * (j1 + 2.0 * j2 + 2.0 * j3 + j4);

  return;
}
```

Further notes

- Augmenting order of method versus augmenting step size
- Adaptive step size
- Other methods

XV Linear Algebra

Introduction

- Storing vectors in memory (row or column major)
- Storing matrices in memory (row or column major)
  - Simulating having a 2D matrix with a preprocessor macro
  - Using an array of pointers to the first element of each row of the matrix
- Reading and printing vectors and matrices

Vectors

Element-wise vector-scalar multiplication

```
void vector_scale(double v[], double vr[], double scalar, int size)
{
  int i;

  for(i = 0; i < size; ++i)
    vr[i] = v[i] * scalar;

  return;
}
```

Element-wise addition

```
void vector_add(double v1[], double v2[], double vr[], int size)
{
  int i;

  for(i = 0; i < size; ++i)
    vr[i] = v1[i] + v2[i];

  return;
}
```

Element-wise subtraction

```
void vector_sub(double v1[], double v2[], double vr[], int size)
{
  int i;

  for(i = 0; i < size; ++i)
    vr[i] = v1[i] - v2[i];

  return;
}
```

Dot product

```
double vector_dot(double v1[], double v2[], int size)
{
  int i;
  double dot = 0.0;

  for(i = 0; i < size; ++i)
    dot += v1[i] * v2[i];

  return dot;
}
```

Vector norm

```
double vector_norm(double v[], int size)
{
  return sqrt(vector_dot(v, v, size));
}
```

Vector cross product

For vectors of size 3 only.

```
void vector_cross(double v1[], double v2[], double vr[])
{
  vr[0] = v1[1] * v2[2] - v1[2] * v2[1];
  vr[1] = v1[2] * v2[0] - v1[0] * v2[2];
  vr[2] = v1[0] * v2[1] - v1[1] * v2[0];

  return;
}
```

Matrices

Element-wise addition

```
void matrix_add(double m1[], double m2[], double mr[],
  int h, int w)
{
  int i, j;

  for(i = 0; i < h; ++i)
    for(j = 0; j < w; ++j)
      mr[in2d(i, j, w)] = m1[in2d(i, j, w)] + m2[in2d(i, j, w)];

  return;
}
```

Element-wise subtraction

```
void matrix_sub(double m1[], double m2[], double mr[],
  int h, int w)
{
  int i, j;

  for(i = 0; i < h; ++i)
    for(j = 0; j < w; ++j)
      mr[in2d(i, j, w)] = m1[in2d(i, j, w)] - m2[in2d(i, j, w)];

  return;
}
```

Element-wise matrix-scalar multiplication

```
void matrix_scale(double m[], double mr[], double scale,
  int h, int w)
{
  int i, j;

  for(i = 0; i < h; ++i)
    for(j = 0; j < w; ++j)
      mr[in2d(i, j, w)] = m[in2d(i, j, w)] * scale;

  return;
}
```

Multiplication

Need `(wm1 == hm2) && (hm1 == hmr) && (wm2 == wmr)`

```
void matrix_mult(double m1[], double m2[], double mr[],
  int hm1, int wm1, int wm2)
{
  int i, j, k;
  double sum;

  for(i = 0; i < hm1; ++i)
    for(j = 0; j < wm2; ++j) {
      sum = 0;
      for(k = 0; k < wm1; ++k)
        sum += m1[in2d(i, k, wm1)] * m2[in2d(k, j, wm2)];

      mr[in2d(i, j, wm2)] = sum;
    }

  return;
}
```

Transposition

```
void matrix_transpose(double m1[], double mr[], int h, int w)
{
  int i, j;

  for(i = 0; i < h; ++i)
    for(j = 0; j < w; ++j)
      mr[in2d(j, i, w)] = m1[in2d(i, j, w)];

  return;
}
```

Gaussian elimination

Introduction

Upper triangularize a matrix equation, then substitute back with the vector.

Gaussian elimination with no pivoting

Needs `m` to be square, cannot have a 0 or a small number on the diagonal.

```
void genp(double m[], double v[], int h, int w)
{
  int row, next_row, col;
  double factor;

  for(row = 0; row < (h - 1); ++row) {
    for(next_row = row + 1; next_row < h; ++next_row) {
      factor = m[in2d(next_row, row, w)] / m[in2d(row, row, w)];

      for(col = 0; col < w; ++col)
        m[in2d(next_row, col, w)] -= factor * m[in2d(row, col, w)];

      v[next_row] -= factor * v[row];
    }
  }

  return;
}
```

Gaussian elimination with partial pivoting (rows only)

Needs `m` to be square.

```
void gepp(double m[], double v[], int h, int w)
{
  int row, next_row, col, max_row;
  double tmp, factor;

  for(row = 0; row < (h - 1); ++row) {

    // Find biggest row.
    max_row = row;
    for(next_row = row + 1; next_row < h; ++next_row)
      if(m[in2d(next_row, row, w)] > m[in2d(max_row, row, w)])
        max_row = next_row;

    // Swap rows.
    if(max_row != row) {
      for(col = 0; col < w; ++col) {
        tmp = m[in2d(row, col, w)];
        m[in2d(row, col, w)] = m[in2d(max_row, col, w)];
        m[in2d(max_row, col, w)] = tmp;
      }
      tmp = v[row];
      v[row] = v[max_row];
      v[max_row] = tmp;
    }

    // Same as in GENP.
    for(next_row = row + 1; next_row < h; ++next_row) {
      factor = m[in2d(next_row, row, w)] / m[in2d(row, row, w)];

      for(col = 0; col < w; ++col)
        m[in2d(next_row, col, w)] -=
          factor * m[in2d(row, col, w)];

      v[next_row] -= factor * v[row];
    }

  }

  return;
}
```

Back substitution

Needs `m` to be square.

```c
void back_substitute(double m[], double v[], int h, int w)
{
  int row, next_row;

  for(row = h - 1; row >= 0; --row) {
    v[row] /= m[in2d(row, row, w)];
    m[in2d(row, row, w)] = 1;
    for(next_row = row - 1; next_row >= 0; --next_row) {
      v[next_row] -= v[row] * m[in2d(next_row, row, w)];
      m[in2d(next_row, row, w)] = 0;
    }
  }

  return;
}
```

APPENDIX A Numbering Systems

The following explanation of the decimal number system might seem very simple and obvious at first, but it is meant to show the parallels between the decimal number system and other bases such as binary, octal and hexadecimal.

We normally use the decimal number system (base 10) and use the following digits to count:

0 1 2 3 4 5 6 7 8 9

When counting in decimal we start from 0 and add 1 each time we want to make a number bigger, thus 0 is followed by 1, then 2, then 3 and so on. When we reach 9, the number that follows will be 10. The leftmost digit is the most significant digit (the one that makes the number bigger) and the rightmost is the least significant. When a digit reaches its last possible value (9 in the decimal number system), its value is reset to zero and the digit just on its right is incremented by one, therefore after 9 comes 10, and after 10 comes 11: we merely increment the least significant digit by 1. From then on we increment the least significant digit until it too reaches the biggest possible digit value, which is 9. When that happens we bring back the least significant digit to 0 and increment the second least significant digit, this is called carrying. When we reach 99 we do as we had done previously: we increment the least significant digit, carry 1 to the second least significant digit, then we carry the second least significant digit to the next least significant digit which in this case happens to also be the most significant digit. This gives us 100 as the following number.

This seems live a fairly trivial and self-obvious explanation because that's what we use in our everyday lives, but there are other numbering systems that exist and they happen to work in exactly the same manner but for one detail: they don't use the same number of digits to count.

Indeed the binary system (base 2) only uses two digits, namely 0 and 1.

The octal system (base 8) uses eight digits:

0 1 2 3 4 5 6 7

And the hexadecimal system (base 16) uses sixteen digits:

0 1 2 3 4 5 6 7 8 9 A B C D E F

Note that in hexadecimal the first six letters of the alphabet are used to represent the digits that follow 9.

As stated before these three numbering systems work in exactly the same manner as the decimal system, for example binary starts counting at 0, which is followed by 1. After this there are no digits of bigger value that binary can work with, therefore we carry on to 10 which is equivalent to 2 in decimal. Then we increment the least significant digit and thus obtain 11, followed by 100, 101, 110, 111, 1000 and so on.

The following table gives the equivalence of numbers in the four above stated bases:

Decimal	0	1	2	3	4	5	6	7
Binary	0	1	10	11	100	101	110	111
Octal	0	1	2	3	4	5	6	7
Hexadecimal	0	1	2	3	4	5	6	7

Decimal	8	9	10	11	12	13	14	15
Binary	1000	1001	1010	1011	1100	1101	1110	1111

Decimal	0	1	2	3	4	5	6	7
Octal	10	11	12	13	14	15	16	17
Hexadecimal	8	9	A	B	C	D	E	F

Decimal	16	17	18	19	20	21	22	23
Binary	10000	10001	10010	10011	10100	10101	10110	10111
Octal	20	21	22	23	24	25	26	27
Hexadecimal	10	11	12	13	14	15	16	17

Decimal	24	25	26	27	28	29	30	31
Binary	11000	11001	11010	11011	11100	11101	11110	11111
Octal	30	31	32	33	34	35	36	37
Hexadecimal	18	19	1A	1B	1C	1D	1E	1F

You should now be able to understand why it's said that:

There are 10 kinds of people, those who understand binary and those who don't.

A nice (undeniably geeky) poem to finish this:

Roses are 0xFF0000
Violets are 0x0000FF
All my base
Are belong to you

Appendix B Data Representation

Integers

Integers are represented in binary on the computer, for example 42 would be represented as `101010`. 32 bit integers (which is what most computers use) always take up 32 bits (no more, no less) thus 42 would be stored as:

```
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 1 0 1 0
```

Some computers might store the bits in the reverse order.
The biggest possible signed 32 bit integer — 2,147,483,647 — is then represented as:

```
0 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
```

When representing negative numbers there is no “sign” bit, instead -1 is:

```
1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
```

-2 is:

```
1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 0
```

-3 is:

```
1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 0 1
```

And so on. The smallest possible signed 32 bit integer — -2,147,483,648 — is then represented as:

```
1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
```

Characters

Characters are represented as integers: a certain number represents each possible characters. In the ASCII standard, which uses 8 bits to represent characters, numbers 0 through 31 represent non-printable control characters, 32 through 126 represent commonly used characters, 127 is another non-printable control character, and 128 through 255 are “extended” characters. The following table contains characters 32 through 126:

32	*space*	33	!	34	"	35	#	36	$	37	%	38	&	39	'
40	(	41	)	42	*	43	+	44	,	45	-	46	.	47	/
48	0	49	1	50	2	51	3	52	4	53	5	54	6	55	7
56	8	57	9	58	:	59	;	60	<	61	=	62	>	63	?
64	@	65	A	66	B	67	C	68	D	69	E	70	F	71	G
72	H	73	I	74	J	75	K	76	L	77	M	78	N	79	O
80	P	81	Q	82	R	83	S	84	T	85	U	86	V	87	W
88	X	89	Y	90	Z	91	[	92	\	93	]	94	^	95	_
96	`	97	a	98	b	99	c	100	d	101	e	102	f	103	g
104	h	105	i	106	j	107	k	108	l	109	m	110	n	111	o
112	p	113	q	114	r	115	s	116	t	117	u	118	v	119	w
120	x	121	y	122	z	123	{	124	|	125	}	126	~		

Floating point numbers

See the following for a very comprehensive explanation:
http://docs.sun.com/source/806-3568/ncg_goldberg.html

APPENDIX C Useful References

The following are a few website where you can find information related to this course and to the material seen in class, and which often offer more information than that seen in class:

- www.wikipedia.org
- www.nist.gov/dads/
- http://www.gnu.org/software/gsl/
- http://mathworld.wolfram.com/
- http://www.nr.com/